Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 3

Алгоритмы с бинарными деревьями

По дисциплине: ООП

Обучающийся: Арройо Николас

Факультет: Факультет инфокоммуникационных технологий

Группа: К3220

Направление подготовки: 11.03.02 Инфокоммуникационные технологии и системы связи

Образовательная программа: Программирование в инфокоммуникационных системах

Обучающийся _____ Арройо Н.
(дата) (подпись) (Ф.И.О.)

Руководитель _____ Иванов С. Е.
(дата) (подпись) (Ф.И.О.)

ЦЕЛЬ РАБОТЫ

Изучить алгоритмы работы с деревьями. Реализовать средствами ООП дерево и рекурсивные функции, выполняющие обход дерева в прямом, обратном, концевом порядке. Вычислить значение выражения, заданного деревом.

Основные теоретические сведения

Деревья являются одной из ключевых структур данных в программировании. Бинарное дерево — это структура, в которой каждый узел имеет не более двух подузлов: левый и правый. Такие деревья широко применяются в компиляторах, вычислениях выражений и системах поиска.

Для обхода бинарного дерева используются различные алгоритмы: прямой (preorder), обратный (inorder) и концевой (postorder).

- В прямом обходе сначала посещается текущий узел, затем левое поддерево и правое.
- В обратном обходе сначала левое поддерево, затем текущий узел и правое.
- В концевом обходе сначала оба поддерева, а в конце — текущий узел.

Также бинарные деревья могут использоваться для вычисления арифметических выражений, если каждый узел содержит операцию или число. Такой подход позволяет интерпретировать и вычислять выражения с учетом приоритетов операций, представленных в виде дерева.

В данной лабораторной работе создаётся бинарное дерево из строки в виде обхода с маркером пустых узлов (.), и реализуются все три вида обхода, а также вычисление значения выражения, заданного в виде дерева.

Задание на лабораторную работу

В данной лабораторной работе необходимо реализовать бинарное дерево средствами ООП и написать рекурсивные функции, выполняющие обход дерева в трёх порядках: прямом (preorder), обратном (inorder) и концевом (postorder). Также требуется вычислить значение арифметического выражения, заданного в виде дерева, используя постфиксный (концевой) обход. В ходе выполнения необходимо:

1. Построить дерево по заданному символьному представлению.
2. Реализовать и протестировать три вида обхода.
3. Рассчитать значение выражения, представленного в виде дерева операций.

Сначала был создан класс Tree, реализующий структуру бинарного дерева. Внутри класса объявлены три поля: Left и Right — ссылки на левого и правого потомков соответственно, а также Value — символ, хранящий значение узла дерева (например, оператор или число). Кроме того, используется статическая переменная list, предназначенная для хранения результата обхода, и index, которая используется при построении дерева из линейного представления. Эта часть структуры представлена на рисунке 1.1.

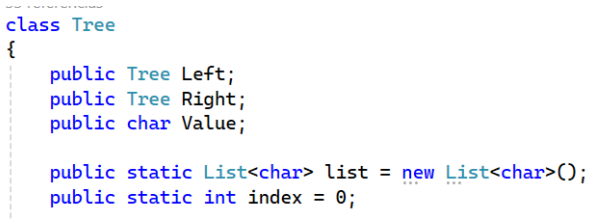

Рисунок 1.1 — Класс Tree: переменные и конструктор

Для инициализации узла был реализован конструктор Tree(char value), принимающий символ и присваивающий его полю Value. Это позволяет создавать дерево из произвольного списка символов, в том числе содержащего пустые элементы (обозначенные точкой '.').

Для построения дерева по линейному представлению используется статический метод Build, показанный на рисунке 1.2. Метод рекурсивно проходит по списку символов и строит дерево, увеличивая index на каждом шаге. Если текущий символ — точка, возвращается null, что означает отсутствие узла. Иначе создаётся новый узел, и рекурсивно строятся его левое и правое поддеревья.

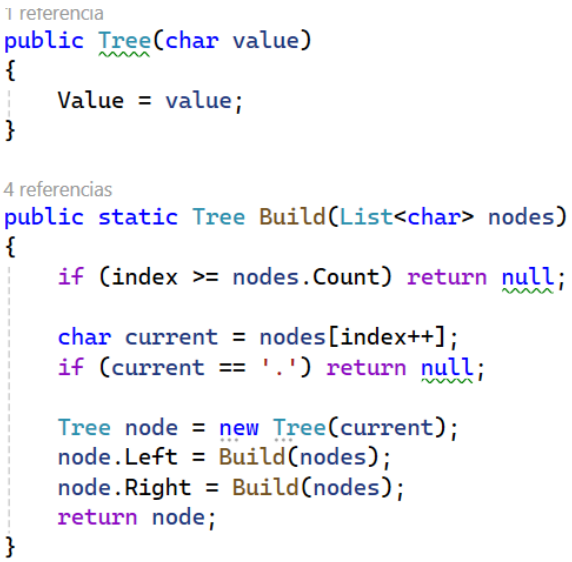

Рисунок 1.2 — Метод Build: построение дерева из списка символов

После построения дерева реализуются три типа обхода: прямой (preorder), обратный (inorder) и концевой (postorder), каждый из которых реализован в виде отдельного метода. Метод TreeWalk_Pr осуществляет прямой обход: сначала добавляется значение текущего узла, затем рекурсивно обходятся левый и правый подузлы. Эта логика показана на рисунке 1.3.

```
public static void TreeWalk_Pr(Tree node)
{
    if (node == null) return;
    list.Add(node.Value);
    TreeWalk_Pr(node.Left);
    TreeWalk_Pr(node.Right);
}
```

Рисунок 1.3 — Метод TreeWalk_Pr: прямой обход дерева

Метод TreeWalk_Obr реализует обратный (inorder) обход: сначала происходит обход левого поддерева, затем добавляется значение текущего узла, и в конце — правого. В отличие от других, этот метод возвращает список List<char>, который затем можно использовать отдельно. Это представлено на рисунке 1.4.

```
public static List<char> TreeWalk_Obr(Tree node)
{
    if (node == null) return new List<char>();
    var result = TreeWalk_Obr(node.Left);
    result.Add(node.Value);
    result.AddRange(TreeWalk_Obr(node.Right));
    return result;
}
```

Рисунок 1.4 — Метод TreeWalk_Obr: обратный обход дерева

Концевой (postorder) обход реализован в методе TreeWalk_K: сначала обходятся левый и правый потомки, и только затем добавляется значение текущего узла в список. Соответствующий код можно увидеть на рисунке 1.5.

```
public static void TreeWalk_K(Tree node)
{
    if (node == null) return;
    TreeWalk_K(node.Left);
    TreeWalk_K(node.Right);
    list.Add(node.Value);
}
```

Рисунок 1.5 — Метод TreeWalk_K: концевой обход дерева

Потом, реализован метод CalcTree, предназначенный для вычисления значения выражения, записанного в виде дерева. Если текущий узел содержит цифру, она преобразуется в число и возвращается. В противном случае рекурсивно вычисляются значения левого и правого поддеревьев, а затем в зависимости от оператора (+, -, *, /) возвращается результат вычисления. Деление также обрабатывается с проверкой деления на ноль. Эта логика представлена на рисунке 1.6.

```
public static int CalcTree(Tree node)
{
    if (node == null) return 0;
    if (char.IsDigit(node.Value)) return node.Value - '0';
    int left = CalcTree(node.Left);
    int right = CalcTree(node.Right);
    return node.Value switch
    {
        '+' => left + right,
        '-' => left - right,
        '*' => left * right,
        '/' => right != 0 ? left / right : 0,
        _ => 0,
    };
}
```

Рисунок 1.6 — Метод CalcTree: вычисление выражения из дерева

Наконец, для наглядного отображения структуры дерева был реализован метод PrintTree, который рекурсивно обходит все узлы дерева и выводит в консоль информацию о каждом из них. В частности, для каждого узла отображается его значение, а также значения левого и правого потомков (или

null, если потомка нет). Метод позволяет удобно проверять корректность построения дерева и визуально представить его структуру. Эта часть показана на рисунке 1.7.

```
public static void PrintTree(Tree node)
{
    if (node == null) return;
    string left = node.Left != null ? node.Left.Value.ToString() : "null";
    string right = node.Right != null ? node.Right.Value.ToString() : "null";
    Console.WriteLine($"value: {node.Value} | left: {left} | right: {right}");
    PrintTree(node.Left);
    PrintTree(node.Right);
}
```

Рисунок 1.7 — Метод PrintTree: вывод структуры дерева в консоль

Основная логика выполнения лабораторной работы реализована в методе Main, как показано на рисунке 1.8. В первой части создаётся список символов chars, представляющий структуру простого символьного дерева (как на рисунке 2 методички). Далее с помощью метода Tree.Build строится дерево, и при помощи PrintTree выводится его структура в консоль.

Затем выполняются три вида обхода дерева: прямой (префиксный), обратный (инфиксный) и концевой (постфиксный). Результаты каждого из обходов сохраняются в список list и выводятся в консоль. Эта часть кода показана на рисунке 1.8.

```
0 referencias
static void Main(string[] args)
{
    List<char> chars = new List<char>() { 'a', 'b', 'd', '.', '.', 'e', '.', '.', 'c', '.', 'f', '.', '.' };
    Tree.index = 0;
    Tree.list.Clear();
    Tree tree1 = Tree.Build(chars);
    Tree.PrintTree(tree1);

    Console.WriteLine("\nStraight order (Preorden):");
    Tree.TreeWalk_Pr(tree1);
    Console.WriteLine(string.Join(" ", Tree.list));

    Tree.list.Clear();
    Console.WriteLine("\nReverse order (Inorden):");
    Tree.list = Tree.TreeWalk_Obr(tree1);
    Console.WriteLine(string.Join(" ", Tree.list));

    Tree.list.Clear();
    Console.WriteLine("\nEnd order (Postorden):");
    Tree.TreeWalk_K(tree1);
    Console.WriteLine(string.Join(" ", Tree.list));
```

Рисунок 1.8 — Основная часть метода Main: построение и обход символьного дерева

Во второй части метода Main создаётся новое дерево, представляющее собой арифметическое выражение. Список expr содержит структуру дерева в постфиксной форме, где цифры и операторы записаны в порядке следования, а символы '.' обозначают пустые узлы. Снова вызывается метод Build, который строит дерево выражения, а затем PrintTree выводит его структуру (как на рисунке 3 методички).

После этого выполняется обход дерева в постфиксном порядке, а затем результат вычисляется с помощью метода CalcTree. Все действия и результат выражения выводятся в консоль. Эта часть показана на рисунке 1.9.

```
List<char> expr = new List<char>() {
    '/', '*', '+', '2', '.', '.', '3', '.', '.', '-', '7', '.', '.', '4', '.', '.', '3', '.', '.'
};
Tree.index = 0;
Tree.list.Clear();
Tree tree2 = Tree.Build(expr);
Tree.PrintTree(tree2);

Console.WriteLine("\nEnd order:");
Tree.TreeWalk_K(tree2);
Console.WriteLine(string.Join(" ", Tree.list));

Console.WriteLine("\nThe equation equals:");
Console.WriteLine(Tree.CalcTree(tree2));
```

Рисунок 1.9 — Построение дерева выражения и вычисление результата

На рисунке 1.10 показан результат построения и обхода дерева, содержащего символьные узлы.

```
value: a | left: b | right: c
value: b | left: d | right: e
value: d | left: null | right: null
value: e | left: null | right: null
value: c | left: null | right: f
value: f | left: null | right: null

Straight order (Preorden):
a b d e c f

Reverse order (Inorden):
d b e a c f

End order (Postorden):
d e b f c a
```

Рисунок 1.10 — Результаты обхода символьного дерева

В верхней части отображается структура дерева. Каждая строка показывает значение текущего узла, а также значения его левого и правого потомков. Например, у узла 'a' левым потомком является 'b', а правым — 'c'. Это дерево соответствует примеру из методички (на рисуноке 2 методички).

Ниже представлены результаты трёх видов обхода:

- В прямом порядке (Preorder): a b d e c f — сначала посещается корень, затем левое поддерево, затем правое;
- В обратном порядке (Inorder): d b e a c f — сначала левое поддерево, затем корень, затем правое;
- В концевом порядке (Postorder): d e b f c a — сначала оба поддерева, затем корень.

На рисунке 1.11 продемонстрирован результат работы с деревом выражения.

```
value: / | left: * | right: 3
value: * | left: + | right: -
value: + | left: 2 | right: 3
value: 2 | left: null | right: null
value: 3 | left: null | right: null
value: - | left: 7 | right: 4
value: 7 | left: null | right: null
value: 4 | left: null | right: null
value: 3 | left: null | right: null

End order:
2 3 + 7 4 - * 3 /

The equation equals:
5
```

Рисунок 1.11 — Вычисление выражения через дерево

Каждый узел дерева содержит либо оператор (+, -, *, /), либо операнд (цифра). Структура дерева построена таким образом, что операторы соединяют подузлы-операнды. Например, корень '/' делит результат выражения из левого поддерева (*) на значение правого узла (3).

В результате выполнения концевого обхода получается постфиксная форма выражения: 2 3 + 7 4 - * 3. Это означает, что сначала складываются 2 и 3, затем результат умножается на разность между 7 и 4, и, наконец, результат делится на 3. Такой способ вычисления основан на принципе обратной польской записи и упрощает обработку без необходимости учитывать приоритеты операций. Этот порядок позволяет последовательно вычислить результат с использованием стека. Итоговое значение выражения, составляет 5.

Контрольные вопросы

1. Какие ограничения и условия на применения алгоритма обхода?

Алгоритмы обхода применимы только к корректно построенным деревьям, в которых каждая вершина может иметь не более двух потомков. Обход неэффективен при циклических структурах и требует, чтобы дерево было полностью доступно в памяти. Кроме того, при рекурсивной реализации возможен стековый переполнение при очень глубоком дереве.

2. К какому классу сложности относится алгоритм обхода?

Алгоритмы обхода дерева (прямой, обратный, концевой) имеют временную сложность O(n), где n — количество узлов в дереве. Каждый узел посещается один раз, что обеспечивает линейную сложность.

3. Какие конструкции ООП C# применяются для реализации прямого, обратного и концевого обхода?

Используются классы (Tree, Program), рекурсия (методы TreeWalk_Pr, TreeWalk_Obr, TreeWalk_K), инкапсуляция данных внутри методов и работа со списками (List<char>). Также применяется статическая переменная index для пошагового построения дерева.

4. Как оценивается сложность алгоритма обхода?

Сложность оценивается по количеству посещений узлов. Так как каждый узел посещается ровно один раз, общая сложность составляет O(n) по времени и O(h) по памяти, где h — высота дерева (глубина рекурсии).

5. Какие задачи решаются с помощью алгоритма обхода?

С помощью алгоритмов обхода можно:

- выводить структуру дерева в разных порядках,
- искать элементы,
- вычислять выражения, заданные деревом (например, арифметические),
- сериализовать/десериализовать дерево,
- копировать и удалять деревья.

6. Как можно оценить точность алгоритма обхода?

Точность можно оценить путем сопоставления результатов обхода с ожидаемыми (например, сравнение с вручную рассчитанным порядком обхода). Также можно использовать тестовые деревья и проверку возвращаемых значений на корректность при вычислении выражений.

7. Какой обход дерева применяется для вычисления выражения?

Для вычисления выражения, заданного бинарным деревом, применяется концевой обход (postorder), так как он обеспечивает правильный порядок выполнения операций в выражении: сначала вычисляются поддеревья (аргументы), затем применяется оператор.

ЗАКЛЮЧЕНИЕ

В процессе выполнения работы я изучил принципы построения и обхода бинарных деревьев, а также реализовал соответствующие алгоритмы средствами объектно-ориентированного программирования на языке C#. Я подробно ознакомился с тремя основными типами обхода дерева: прямым (preorder), обратным (inorder) и концевым (postorder), а также научился применять их для анализа структуры дерева и выполнения вычислений.

На практике был создан класс Tree, инкапсулирующий структуру узла и реализующий методы построения дерева, его обхода и вычисления арифметического выражения. Построение дерева происходило из линейного списка символов с использованием рекурсии и статической переменной index, а вычисление выражения осуществлялось через постфиксный обход, что позволило корректно обрабатывать приоритеты операций.

Работа над проектом позволила закрепить навыки работы с классами, рекурсией, списками и обработкой символьных структур. Я также убедился в том, насколько важна правильная организация кода и использование ООП для реализации структур данных и алгоритмов обработки.